BINARY DECISION DIAGRAM

DAVID ALEXANDRE

JOSÉ SILVINO NETO

PEDRO HENRIQUE

https://github.com/Mirajan/Huffman-Project

MOTIVATION

- Boolean Functions are mostly represented by truth tables and propositional formulas

P	Q	$P \wedge Q$	$P \vee Q$	$P \underline{\vee} Q$	$P \underline{\wedge} Q$	$P \Rightarrow Q$	$P \Leftarrow Q$	$P \Leftrightarrow Q$
T	T	T	T	F	T	T	T	T
T	F	F	T	T	F	F	T	F
F	T	F	T	T	F	T	F	F
F	F	F	F	F	T	T	T	T

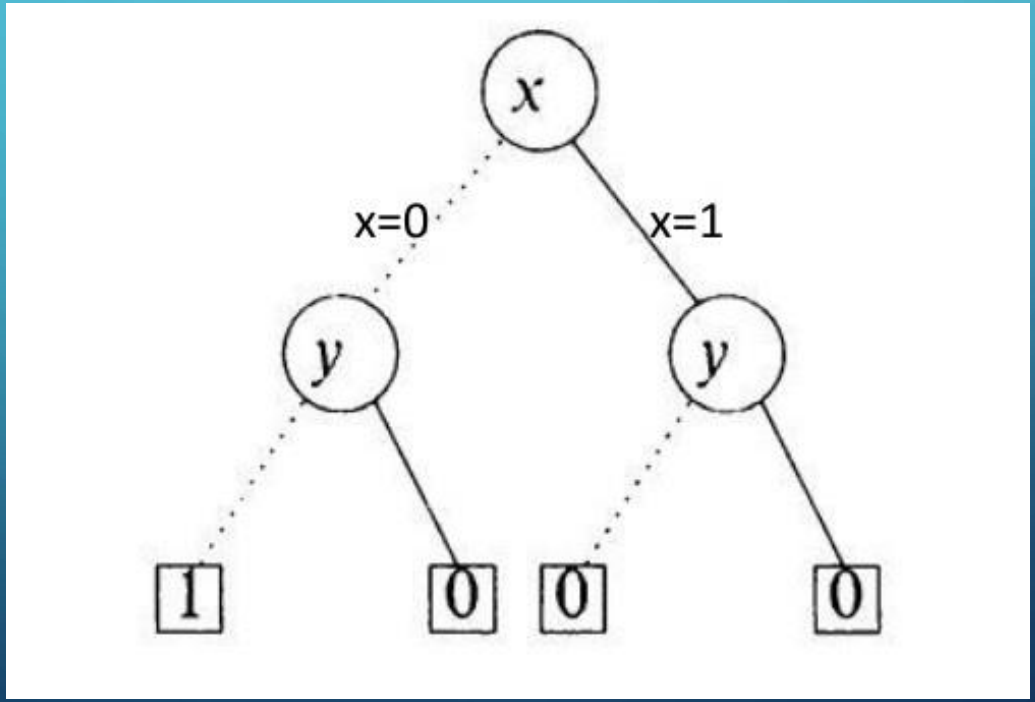
HOWEVER, THERE ARE SOME PROBLEMS WITH THESE REPRESENTATIONS...

- REPRESENTATION WITH TRUTH TABLE
  - Space inefficient: 100 variables results in 2^{100} lines.
  - Checking satisfiability and equivalence is also inefficient.

- REPRESENTATION WITH PROPOSITIONAL
  - Better than Truth Table in terms of compact representation.
  - Deciding if two propositional formulas denote the same function is expansive.

➢ Another way of representing Boolean functions: BDD

- BDD's are trees with their leaves marked 0 or 1 and all other nodes marked with Boolean variables.
- Each node has two edges.



DEFINITIONS

- Nodes

- Nodes represent Boolean variables.
- In this structure instead of having a structure for the separate node, we will have a unique structure for the BDD and the node, that is, each node can be considered a BDD starting at that node.

CODE

- The following structure will represent the BDD:

```
struct BDD
{
    int index;
    BDD *high;
    BDD *low;
    BDD *next;
};
```

- To ensure that the nodes are not identical and maintain the canonicity:

```
#define PAIR(a,b) ((unsigned int)((((unsigned int)a) + ((unsigned int)b)) *
   (((unsigned int)a) + ((unsigned int)b) + ((unsigned int)1))/((unsigned int)2) +
   ((unsigned int)a)))

#define TRIPLE(a, b, c) ((unsigned int)(PAIR((unsigned int)c, PAIR(a, b))))

#define NODEHASH(lvl, l, h) (TRIPLE((lvl), (l), (h)) % BDDnodesize);
```

Functions

- Make Node
- Ithvar
- Restrict
- ITE
- Satcount
- Satone

- The make Node function part 1

```
unsigned int hash = NODEHASH(var, high, low);
 if (BDDTable[hash] == 0) {
     BDDTable[hash] = new BDD(var, high, low);
     return BDDTable[hash];
 }
}
```

- The Make Node Function part 2: The age of Ultron

```
else {
        BDD *node = BDDTable[hash];
        while (node -> next &&(node -> index != var || node -> high != high || node -> low != low))
            node = node -> next;
        if (node -> index != var || node -> high != high || node -> low != low) {
            BDD* newNode = new BDD(var, high, low);
            node -> next = newNode;
            return newNode;
        }else
            return node;
    }
```

- Ithvar

```
BDD *Ithvar(int i)
{
    numvars ++;
    BDDTable[0] -> index = numvars;
    BDDTable[1] -> index = numvars;
    return Make_Node(i, BDD_True(), BDD_False());
}
```

- Restrict

```
BDD *BDD_Restrict(BDD* subtree, int var, bool val)
{
    if (subtree -> index > var) {
        return subtree;
    }else if (subtree -> index < var) {
        return Make_Node(subtree -> index, BDD_Restrict(subtree -> high, var, val), BDD_Restrict(subtree ->
    low, var, val));
    }else {
        if (val)
            return BDD_Restrict(subtree -> high, var, val);
        else
            return BDD_Restrict(subtree -> low, var, val);
    }
}
```

- ITE part 1

```
BDD *ITE(BDD *I, BDD *T, BDD *E) {
    if (I == BDD_True()) return T;
    if (I == BDD_False()) return E;
    if (T == E)
    if (T == BDD_True() && E == BDD_False()) return I;
    int split_var  = I -> index;
    if (split_var > T -> index)
        split_var = T -> index;
    if (split_var > E -> index)
        split_var = E -> index;
```

- ITE part 2: A Clash of Kings

```
BDD *Ixt = BDD_Restrict(I, split_var, true);
    BDD *Txt = BDD_Restrict(T, split_var, true);
    BDD *Ext = BDD_Restrict(E, split_var, true);
    BDD *pos_ftor = ITE(Ixt, Txt, Ext);
    BDD *Ixf = BDD_Restrict(I, split_var, false);
    BDD *Txt = BDD_Restrict(T, split_var, false);
    BDD *Exf = BDD_Restrict(E, split_var, false);
    BDD *neg_ftor = ITE(Ixf, Txf, Exf);
    BDD *result = Make_Node(split_var, pos_ftor, neg_ftor);
    return result; }
```

- Satcount part 1

```
double BDD_Satcount_Rec(BDD *subtree)
{
    if (subtree == BDD_False()) {
        return 0;
    }
    else if (subtree == BDD_True()){
        return 1;
    }
```

- Satcount part 2: Attack of the Clones

```
else{
    BDD *low = subtree -> low;
    BDD *high = subtree -> high;
    double size = 0.0, s = 1.0;
    s = pow(2.0, (low -> index) - (subtree -> index) - 1);
    size += s * BDD_Satcount_Rec(low);
    s = pow(2.0, (high -> index) - (subtree -> index) - 1);
    size += s * BDD_Satcount_Rec(high);
    return size;
  }
}
```

- Satcount part 3: Revenge of the Sith

```
double BDD_Satcount(BDD *subtree)
{
    return pow(2.0, (subtree -> index) - 1) * BDD_Satcount_Rec(subtree);
}
```

- Satone Rec

```
assert(subtree != 0);
    if (subtree == BDD_False() | | subtree == BDD_True()){
        return subtree;
    } else if (subtree -> low == BDD_False()){
        return Make_Node(subtree -> index, BDD_False(), BDD_Satone_Rec(subtree -> high));
    } else{
        return Make_Node(subtree -> index, BDD_Satone_Rec(subtree -> low), BDD_False());
    }
```

- Satone

```
assert(subtree != 0);
    if (subtree == BDD_False())
        return 0;
    else
        return BDD_Satone_Rec(subtree);
```

BENEFITS

- BDDs can be much more practical and efficient than tables of truths or propositional formulas.
- They are simpler to visualize.
- They are the best way to represent Boolean expressions.

CONCLUSION

- What was presented on this slide is a highly simplified pointer-based implementation of a BDD. For the sake of simplicity, things like error checking and data collection, as well as many efficiency considerations, were omitted implementation.

QUESTIONS?

THANKS!